DS d'informatique n°2 : Suite de Champernowme

Un exercice indépendant

On considère l'algorithme suivant :

```
Entrée : a,b
# Retourne pgcd(a,b), avec a,b deux entiers positifs et (a,b)≠(0,0)
Tant que a*b > 0 faire :
    Si a > b :
        a ← a-b
    Sinon :
        b ← b-a
Retourner le maximum entre a et b
```

/ 4 **1)** On notera a_k et b_k les valeurs des variables a et b à la fin de la k-ième itération de la boucle "Tant que".

k	a_k	b_k
0	72	30
1	42	30
2	12	30
3	12	18
4	12	6
5	6	6
6	6	0

La valeur retournée est le maximum entre 6 et 0, donc 6.

/ 11 **2)** Le seul obstacle à la terminaison est la boucle "Tant que". On pose $v_k = a_k b_k$. Montrons que v_k est un variant de boucle.

Par récurrence, montrons que $a_k, b_k \in \mathbb{N}$. Pour $k = 0$, c'est le cas par hypothèe : $a_0 = a$ et $b_0 = b$ sont dans $\mathbb{N}$. De plus, si $a_k, b_k \in \mathbb{N}$, alors :

- Si $a_k > b_k$, on a $b_{k+1} = b_k \in \mathbb{N}$ et $a_{k+1} = a_k - b_k \in \mathbb{N}$.
- De même, si $a_k \leq b_k$, on a bien $a_{k+1}, b_{k+1} \in \mathbb{N}$.

Finalement, on a bien $a_k, b_k \in \mathbb{N}$ pour tout k. Comme $v_k = a_k b_k$, on en déduit que $\boxed{v_k \text{ est un entier}}$ et que $\boxed{v_k \geq 0}$.

- Montrons que (v_k) est strictement décroissante. Pour tout k, si l'itération $k+1$ a lieu, cela implique que nécessairement $a_k b_k > 0$, donc $a_k > 0$ et $b_k > 0$ (car ces deux quantités sont positives par ce qui précède). Ainsi :
  - Si $a_k > b_k$, alors $a_{k+1} = a_k - b_k < a_k$ et $b_{k+1} = b_k > 0$. On a alors $a_{k+1} b_{k+1} < a_k b_k$. Donc $v_{k+1} < v_k$.
  - Si $a_k \leq b_k$, alors on montre de même que $v_{k+1} < v_k$.

  Dans tous les cas, on a bien $\boxed{v_{k+1} < v_k}$: c'est une suite strictement décroissante.

Finalement, v_k est bien un variant de boucle : la boucle "Tant que" termine, de même que l'algorithme.

Partie A : suite de Champernowme

/ 6 **1)**

```
1  champ(N : int) -> str :
2      S=''  # chaine vide
3      for k in range(N+1):     # de 0 à N inclus donc range(N+1)
4          S = S + str(k)
5      return S
```

/ 10 **2)**

```
1  champ2(N : int) -> str :
2      S = champ(N+1)
3      # On ajoute à S les chiffres des nombres de 0 à N. Comme
           chacun de ces nombres contient au moins un caractère, S
           contient au moins N+1 caractères
4      return S[0:N+1] # on prend les N+1 premiers caractères de S
           (de 0 inclus à N+1 exclu).
```

Partie B : suite de Champernowme réduite, algorithme naïf

/ 10 **3)**

```
1  def cherche(mot:str, texte:str) -> bool :
2      n = len(texte)
3      m = len(mot)
4      for k in range(n):
5          S = texte[k:k+m] # sous-chaine de m caractères depuis le
               caractère numéro k de texte, cela ne pose pas de
               problème si k+m > n : texte sera alors de longueur < m
               donc différent de mot
6
7          if S == mot: # comparaison entre chaines, cf annexe
8              return True
9      return False
```

/ 7 **4)** Le pire cas arrive lorsqu'on cherche un mot qui n'est pas dans le texte. Les opérations élémentaires sont :

- Le "−" ligne 4, répété 1 fois
- Le "+" ligne 5, répété n fois
- Le "==" ligne 6, répété n fois.

Finalement, on effectue $2n+1$ opérations. On obtient ainsi une complexité d'ordre n, donc linéaire. *Note : d'autres algorithmes peuvent aboutir à (par exemple)* $2(n-m)+1$ *opérations élémentaires. On considère alors* m *petit devant* n *pour conclure.*

/ 8 **5)**

```
1  def champred(N : int) -> str :
2      C = ''
3      for k in range(N+1) :
4          if cherche( str(k) , C ) == False : # si l'entier k n'est
               pas déjà dans C
5              C = C + str(k)  # on l'ajoute à C
6      return C
```

/ ?

6) (Barème variable) On remarque que dans la ligne 4, l'instruction `cherche( str(k), C )` est répétée $N+1$ fois et comme sa complexité est linéaire en la taille de `C`, elle réalise donc un nombre d'opérations de $2n+1$ avec n la taille de `C`. Or cette taille varie de 0 à la taille finale de C qu'on peut noter $t_C(N)$. Or, $t_C(N)$ est très difficile à déterminer ou estimer en fonction de N... On peut en première approximation supposer que $t_C(N)$ est proportionnelle à N d'après les exemples, sans pour autant le justifier. Faisons cette hypothèse.

De 0 à $t_C(N)$, la taille moyenne de C est donc environ $n_{moy} = \dfrac{t_C(N)}{2}$, et donc `cherche( str(k), C )` réalisera en moyenne $2n_{moy}+1 = t_C(N)+1 \approx N+1$ opérations élémentaires. Comme on répète cette instruction $N+1$ fois, on réalisera ainsi $(N+1)^2$ opérations élémentaires, donc au moins N^2.

Partie C : suite de Champernowme réduite, deuxième algorithme

/ 5

7) Si N est de longueur ℓ, alors on ajoutera à la chaine C_r uniquement des chaines correspondant à des entiers $k \leq N$, qui sont donc de longueur inférieure ou égale à ℓ. Il suffit donc de stocker dans `X` des chaines de longueurs inférieures ou égales à ℓ.

/ 5

8) Étant donné que la comparaison entre deux chaines de caractères est considéré (dans ce devoir) comme une opération élémentaire, si on cherche à savoir si une chaine `S` est présente dans `L`, il suffit de parcourir chaque élément de `L` et de tester `L[i]==S` pour chaque indice `i` : cela conduit à une opération élémentaire par élément de `L`, donc un coût linéaire en la taille de `L`.

Dans un dictionnaire `D` (solution b), si on cherche à savoir si une chaine `S` est présente dans `D`, il suffit de voir si `D[S]` retourne `True` ou `False`. Cela constitue une opération élémentaire (cf annexe). Ainsi, le coût est constant.

/ 5

9) On peut construire un dictionnaire dont les clés sont des chaines ou des entiers, les deux sont acceptés.

```
1 def initdico(N:int) -> dict :
2     D={} # dico vide, sinon D={'0':False} ou D={0:False} OK
3
4     for k in range(N+1): # on va de 0 à N inclus
5         D[str(k)]= False      # création d'une nouvelle clé
6         # D[k] était accepté aussi
7     return D
```

/ 5

10)

```
1 def longueur(N:int) -> int:
2     return len(str(N)) # str(124) -> '124' qui est de longueur 3
```

/ 8

11) $N \in \mathbb{N}^*$ s'écrit avec M chiffres si et seulement si

$$10^{M-1} \leq N < 10^M$$

et donc en passant au logarithme dans la première inégalité, on a $(M-1)\ln 10 \leq \ln N$. D'où $M \leq \dfrac{1}{\ln 10}\ln N + 1$.

- Si $N = 1$, on a $M = 1$ mais $\ln N = 0$: il est donc impossible d'avoir $M \leq \alpha \ln N$ pour toute constante α (certains l'ont remarqué et cela a été valorisé).
- Si $N \geq 2$, on a alors

$$M \leq \ln N \left(\frac{1}{\ln 10} + \frac{1}{\ln N}\right) \leq \ln N \left(\frac{1}{\ln 10} + \frac{1}{\ln 2}\right)$$

donc en posant $\alpha = \dfrac{1}{\ln 10} + \dfrac{1}{\ln 2}$, on a le résultat.

/ 10

12)

```
1  def champred2(N : int)-> str :
2      M = longueur(N)
3      C = ''
4      D = initdico(10**M) # on doit pouvoir stocker toutes les
           chaines qui s'écrivent avec max. M chiffres
5      for k in range(N+1) :
6          if D[str(k)] == False : # D[k] est OK en adaptant la l. 10
7              for chiffre in str(k):
8                  C = C + chiffre # ajout de chiffre dans C
9                  for j in range(1,M+1):
10                     # Déclarer la séquence des j derniers
                           caractères de C comme déjà vues
11                     D[ C[-j:] ] = True
12     return C
```

/ 10

13) Majorons le nombre d'opérations élémentaires de `champred2(N)`. On va considérer que la condition `D[k]==False` ligne 6 est toujours vérifiée, de sorte que toutes les opérations élémentaires effectuées ensuite sont toujours exécutées (c'est un pire cas "encore pire" théorique). La majoration du nombre d'opérations élémentaires effectuées sera alors :

a) Ligne 2 : `longueur(N)` réalise un nombre d'opérations élémentaires qui dépend des conventions qu'on se donne. La convention la plus pessimiste dénombre M opérations élémentaires pour convertir `str(N)` puis une pour appliquer la fonction `len`. Donc au plus $\boxed{M+1}$ opérations élémentaires.

b) Ligne 4 : l'instruction `10**M` réalise $\boxed{1}$ opération élémentaire.

c) Ligne 4 : `initdico(10**M)` réalise $\boxed{10^M + 2}$ opérations élémentaires car la création d'une nouvelle clé constitue une opération élémentaire (rajouté dans l'énoncé dans le DS).

d) Ligne 5 : le $+$ est exécuté 1 fois, soit $\boxed{1}$ opération élémentaire.

e) Ligne 6 : le $==$ est exécuté $N+1$ fois, soit un total de $\boxed{N+1}$ opérations élémentaires.

f) Ligne 8 : le $+$ constitue une opération élémentaire ici, et est répété au plus $(N+1)M$ fois, car `str(k)` contient au plus M éléments. Donc au plus $\boxed{(N+1)M}$ opérations élémentaires.

g) Ligne 9 : le $+$ constitue, de même, au plus $\boxed{(N+1)M}$ opérations élémentaires

h) Ligne 10 : `C[-j:]` constitue une opération élémentaire (le signe `-` devant `j`), et elle est répétée $(N+1)M \times M$ fois, donc $\boxed{(N+1)M^2}$ opérations élémentaires.

i) Ligne 10 : la modification du dictionnaire `D` constitue une opération élémentaire (cf annexe), donc comme ce qui précède, cela donne $\boxed{(N+1)M^2}$ opérations élémentaires.

Ainsi, on voit que pour chacune de ces 9 opérations, le nombre d'opérations réalisé est majoré par $2NM^2$ (si on suppose $N \geq 1$). Ainsi on peut dire (la majoration est très grossière) que le nombre total est majoré par $9 \times 2NM^2 = \boxed{18N(\ln N)^2}$.